Garbage Collection

15

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Java memory management.
- Garbage collection.
- How to code according to the garbage collector.
- How to make objects eligible for collection.
- The latest updates of garbage collection.

15.1 | Introduction

Memory management is always an extremely important part of any program or application being developed in the Java programming language. Whenever objects are created for classes in any programs or applications developed using Java language, they are allocated a certain amount of memory, allowing the program or application to function the way that it should.

Just like objects in Java, variables are also allocated memory that allows them to be stored and used throughout the program. All variables and objects need to be assigned or allocated certain areas of memory to ensure that they can be used without errors or exceptions manifesting themselves. However, the memory that is allocated to them differs according to the scope that the said variable is located in.

Once the memory allocated to a certain variable or object is deemed useless and the purpose of the variable or object has been completed, it is important for the allocated memory to be recycled and be usable for other purposes. And that is where the garbage collector in Java comes into play.

Fortunately for developers of the Java programming language, memory reclamation is automatic in the Java virtual machine (JVM), which means that Java developers do not necessarily have to go out of their way in order to free memory objects that are no longer being used. Garbage collection in Java also works based on the assumption that objects are short-lived and can be easily reclaimed once they have been created.

Another plus in Java is that if there are ever any objects that are not referenced, they are automatically removed from heap memory to free up space for other objects and variables, making this an extremely memory-efficient language.

What will happen to your program at runtime if there is no Garbage Collection taking place?

15.2 | Garbage Collection in Java

Since garbage collection has to do with memory management more than anything else, it is imperative for us to talk about how memory works in Java programming. It is clear that garbage collection in Java is an automatic process, and that the programmer or developer creating a piece of code or developing an application does not need to go out of their way in order to tell the machine which objects in the code need to be deleted. However, there are certain specifications that need to be met to ensure that the object will be deleted. And this is where memory management is important.

For the sake of simplification, let us say that the memory heap for Java is divided into three major sections – Young Generation, Old (or Tenured) Generation, and Permanent Generation.

Whenever a new object is created, it is located in the Young Generation. This Young Generation is further divided into two subcategories – Eden Space and Survivor Space. Upon creation, new objects are present in the Eden Space and are moved to Survivor Space 1 after the first garbage collection cycle.

A minor garbage collection event then follows in order to move objects from the Young Generation to the Old Generation. The Old Generation contains all objects that have matured enough to be moved from here but cannot fall in the category of Permanent Generation. When the garbage collector needs to remove objects from the Old Generation, this is known as major garbage collection event.

All data that is required for proper functioning of the application or code is stored in the Permanent Generation. This generation, therefore, stores meta data and information regarding classes. In case there is a class that does not need to be used, it can be removed from the Permanent Generation with the help of a garbage collector in order to free memory for other classes or data that is imperative for the code to run properly.

Most developers regard the Permanent Generation as a block that is contained in the native memory instead of the heap memory. Since the Permanent Generation contains class definitions by class loaders, this block has inherently been designed to expand and grow to ensure that there are no out of memory errors or exceptions that are thrown in the code. However, in case the block needs more memory than available in the physical memory, the operating system ensures that virtual memory is made available for the code to run like it should. This virtual memory will certainly allow the code to run; however, in order to make use of this virtual memory, the constant back and forth will be required between the virtual memory and physical memory. This affects the performance and smoothness of the code.

Now that you have a basic understanding of memory heap and how it works, we can start talking about the process that is involved in garbage collection. A daemon thread is created and used by the JVM for garbage collection. Whenever a new object is created, the JVM attempts to get the space that is required for the object from the Eden Space. As is the rule, the Survivor Spaces and Tenured Space are empty at the beginning of the code.

In case the JVM is unable to find the required memory from Eden Space, minor garbage collection is initiated to free up the required space. For this process, one of the two Survivor Spaces, S0 or S1, are regarded as the To Space. Next, all objects that are not reachable are copied by the JVM to the To Space, and 1 is added to their age. On the other hand, all objects that are not fit for the Survivor Space are moved to Tenure Space.

Since not every object is meant to move from the Young Generation Space to the Tenured Space, JVM comes with a Max Tenuring Threshold. This is basically an option that can be modified according to the preferences of the programmer or the requirements of the application to ensure that there is always enough memory for the creation and initiation of new objects. While the default value of the Max Tenuring Threshold is set as 15, it can be changed.

As mentioned earlier, a minor garbage collection process occurs in order to reclaim memory that can be freed from the Young Generation (when objects become mature and move on to Tenured Space). It is important to note that garbage collection is a Stop The World process in Java, which means that the garbage collector ensures that all the threads that are being used to run the application or program are stopped and only the threads that are being used for garbage collection are still running until the process is complete. It is also important to keep in mind that Stop The World will occur regardless of the algorithm that is being used for garbage collection.

The number of threads being used for garbage collection will depend on the algorithm that is being used for the process. Based on the algorithm, garbage collection could either be done successfully using a single thread or multiple different threads working together to clean out memory. Additionally, while the delay caused by the STOP-THE-WORLD application is often negligible, in cases where there is a lot of memory to be cleaned, garbage collector tuning can also be applied to reduce the STOP-THE-WORLD time.

Can we guarantee garbage collection?

15.3 | Major Garbage Collection

If minor garbage collection occurs very frequently, the objects from the Young Generation will naturally move into Tenured Space and occupy all of the available memory very quickly. Since that will prove to be detrimental to the program or application, JVM will trigger a major garbage collection event. While major garbage collection is also referred to as full garbage collection at times, it should be noted that full garbage collection entails reclaiming memory from the Meta Space as well.

And while this is one way in which major garbage collection can be triggered, there are a number of other possibilities as a result of which JVM can call major garbage collection. Even though it is generally advised against, if a programmer decides to

call `Runtime.getRunTime().gc()` or `System.gc()`, the JVM will trigger a major garbage collection. It is also possible for major garbage collection to be triggered if there is not enough memory remaining in the Tenured Space, if the JVM is unable to reclaim the required amount of memory from the Eden Spaces or Survivor Spaces, or if enough space is not available for the JVM to load new classes or objects as they are created in the program or application.

Is there any situation where the garbage collector stops working?

15.4 | G1 and CMS Garbage Collectors

Java offers various different types of garbage collectors, which have their own advantages and disadvantages. It is important to learn about garbage collectors to understand which one to use for your specific needs. The following are the two most significant garbage collector options offered by Java:

1. **Garbage First (G1) garbage collector:** Introduced in Java 7, G1 is capable to handle very large heaps efficiently and concurrently. For Java 9, this is the default garbage collector. If you are using a version prior to Java 9, you may enable G1 with `-XX:+UseG1GC` parameter for JVM. G1 offers various advantages:
   - **(a)** Uncommiting unused heap.
   - **(b)** Free up memory space without using a long pause time.
   - **(c)** Work concurrently such as without interrupting or stopping application threads.
   - **(d)** Deal with very large heap by using non-continuous spaces.
   - **(e)** It can collect both young and old generation spaces at once. This can be achieved by G1 by splitting the heap into hundreds of small regions instead of only three (i.e., Eden, Survivor, and Old) like most other garbage collectors do.

   Mainly, GC1 outshines other garbage collectors on large amount of data as it does not have to work on the entire heap or entire generation. It can simply work on the selected small regions and finish quickly. On the disadvantage side, GC1 struggles to work with small heaps.
2. **Concurrent Mark Sweep (CMS) garbage collector:** As the name suggests, Concurrent Mark Sweep uses multiple threads ("Concurrent"), which are used to scan through the heap and mark the unused objects ("Mark") that can be collected and recycled ("Sweep"). Many applications strive for shorter garbage collection pauses and do not get affected by sharing their processor resources with the garbage collector while the application is running. These are the perfect candidates to use CMS. Also, the benefits of this garbage collector are for the applications which got a large set of long-lived data (such as a large tenured generation) and execute on multiprocessors. You can enable the CMS collector with the following command-line option:

```
-XX:+UseConcMarkSweepGC
```

There are a few challenges in using CMS collector, such as finding the right time to initiate the concurrent work, as this work can get completed before the application is out of available heap space. CMS requires higher percentage of heap space than Parallel garbage collector in a scale of 10% to 20%. Hence, it is a costlier proposition for using shorter garbage collector pause times. Another challenge is related to handling the fragmentation in the old generation. When old generation goes through the garbage collector process, it may occur that the free space between objects get smaller or nearly non-existent. Hence, the objects which are getting promoted from the young generation do not find sufficient place to fit. Since CMS concurrent collection does not do any type of compaction, whether incremental or partial. This unavailability of space for promoted objects forces CMS to a full collection using Serial garbage collector. This results in a lengthy pause.

QUICK CHALLENGE

Write a memory-intensive program which creates a lot of objects. Try G1 and CMS collector on this program. Print timestamp and heap size. Use the following commands to print the heap size and free space.

Command to print total memory of heap:

```
Runtime.getRuntime().totalMemory()
```

Command to print free memory of heap:

```
Runtime.getRuntime().freeMemory();
```

15.5 | Advantages of Garbage Collection in Java

Garbage collection in Java is essential for its benefits pertaining to freeing up memory to ensure that it can be used for other purposes. However, limiting the advantages of garbage collection to just that is an injustice of the highest degree. Since you are not responsible for keeping tabs on the data and figuring out when it is no longer necessary for a certain object or needs to be cleaned after being left behind by an object that is not referenced anymore, you do not only have the time to deal with everything else that is on your plate, but automation of the garbage collection process also makes you more productive.

And that is not all. Since manual garbage collection in Java has been made extremely difficult, it is possible for programmers or developers to accidentally cause the program or application that they are working on to crash due to incorrect removal of certain objects from memory. Additionally, since removal and updating of memory works automatically in the programming language, integrity of the program is maintained at all times.

15.6 | Making Objects Eligible for Garbage Collection

The only reason why an object will be suitable for garbage collection in Java is if the reference variable of the object is no longer available. Objects that fall under this category are also termed as *unreachable objects*.

While there are a number of different ways in which an object can be made eligible for garbage collection, it is important to first talk about the reference of an object and how referencing works. By now, you are probably aware that every object is assigned a certain amount of memory from the available heap, but the operation behind the process is slightly complicated. Every time an object is automatically assigned memory through the operator, a reference to that memory is returned back. For ease of understanding, this reference is basically the address of the newly created object in the memory that is given to it by the "new" operator.

At this point, it is important to note that if an object is being referenced, it is not necessary that it is being used at that particular moment in the program or application. What this means is that if an object is passed as an argument or assigned to a variable, this action essentially only takes the reference of the object into account and does nothing more. An example of this type of referencing is as follows:

```
Book myJavaBook = new Book() ;
```

15.6.1 Unreachable Objects

As mentioned earlier, only objects that no longer have a reference associated with them can be deemed or made eligible for garbage collection. Why is that the case? This is because an object that does not have a reference is essentially not present on the memory heap, which means that it is not available for use and cannot add any value. Figure 15.1 shows the unreachable and reachable objects.

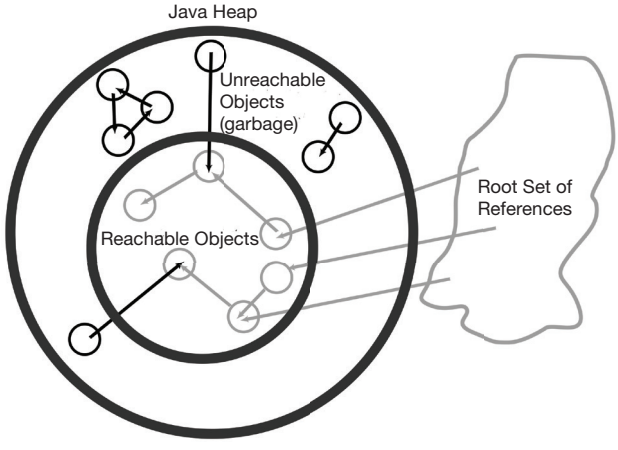

Figure 15.1 Reachable and unreachable objects in Java heap.

Since we are talking about how objects can be made eligible for garbage collection, the first scenario where this process can be applied is when objects are created within the scope of a method. Whenever a method is called, it is pushed on or moves directly to the stack that contains all methods that are important for the successful execution of the program or application. Now, when this method is popped or removed from the stack, all of the members that were associated with this method die with it.

In case there were any objects that were created in this method, they will also die off, leaving unreferenced objects on the heap that can no longer be used for any sort of value addition. Based on the premise that they do not have any reference, these anonymous objects automatically become eligible for garbage collection.

Here is a program to demonstrate that objects that are created within the scope of a method will be deemed useless after execution of that method is complete.

```
package java11.fundamentals.chapter15;
public class UnreachableObjectsExample {
    private String myObject;
    public static void main(String args[])
    {
        // Executing testMethod1 method
           testMethod1();
        // Requesting garbage collection
           System.gc();
    }
    public UnreachableObjectsExample(String myObject)
    {
        this.myObject = myObject;
    }
    private static void testMethod1()
    {
        // After existing testMethod1(), the object myObjectTest1 becomes unreachable
        UnreachableObjectsExample myObjectTest1 = new
        UnreachableObjectsExample("myObjectTest1");
        testMethod2();
    }
    private static void testMethod2()
    {
        // After existing testMethod2(), the object myObjectTest2 becomes unreachable
        UnreachableObjectsExample myObjectTest2 = new
        UnreachableObjectsExample("myObjectTest2");
    }
    @Override
    protected void finalize() throws Throwable
    {
        // following line will confirm the garbage collected method name
        System.out.println("Garbage collection is succcessful for " + this.myObject);
    }
}
```

Since both the objects within the method had become unreachable, the output will be as follows.

```
Garbage collection is succcessful for myObjectTest2
Garbage collection is succcessful for myObjectTest1
```

The output shows that any object within a method becomes useless after execution of the method; the object automatically becomes eligible for garbage collection.

15.6.2 Reassigning Reference Variables

Reference IDs are extremely important in Java and help in addressing each of the objects and variables that are being used in any code. In case, one object's reference ID is used to refer to other object's reference ID, then the first object that was initially being referenced becomes unreachable and cannot be used in the program or code in any way. Once it becomes unreachable due to the reference ID being used for multiple objects, the first object is deemed eligible for garbage collection.

The following program is an example of a situation where a reference ID is used to reference multiple objects.

```
package java11.fundamentals.chapter15;
public class ReassigningReferenceExample {
private String myObject;
    public ReassigningReferenceExample(String myObject)
    {
        this.myObject = myObject;
    }
    public static void main(String args[])
    {
        ReassigningReferenceExample testObject1 = new
        ReassigningReferenceExample("testObject1");
        ReassigningReferenceExample testObject2 = new
        ReassigningReferenceExample("testObject2");
        // testObject1 now refers to testObject2
        testObject1 = testObject2;
        // Requesting garbage collection
        System.gc();
    }
    @Override
    protected void finalize() throws Throwable
    {
        // following line will confirm the garbage collected method name
        System.out.println("Garbage collection is succcessful for " + this.myObject);
    }
}
```

Since the reference ID of the first object, `testObject1`, is eventually being used to reference the second object, `testObject2`, the first object becomes unreachable and is suitable for garbage collection, as shown in the code above. The output of the code will be as follows.

```
Garbage collection is succcessful for testObject1
```

The example above shows the importance of using the right reference ID at the right time to ensure that objects or variables that are crucial for the successful execution of your code, application, or program are not uselessly lost.

15.6.3 Nullified Reference Variables

Another extremely effective method to make an object suitable for garbage collection is making all of the variables that reference to it NULL. When this is done, you will have a scenario similar to the one mentioned above, and the object will have no references to it, essentially making it useless or unreachable. As soon as the object becomes unreachable, it is suitable for garbage collection, and the garbage collector can be called to remove it from the heap.

The following is an example code that shows how nullifying the reference variables of an object can make it unreachable and eligible for garbage collection:

```
package java11.fundamentals.chapter15;
public class NullifiedReferenceVariablesExample {
    private String myObject;
    public NullifiedReferenceVariablesExample(String myObject)
    {
        this.myObject = myObject;
    }
    public static void main(String args[])
    {
        NullifiedReferenceVariablesExample testObject1 = new
        NullifiedReferenceVariablesExample("testObject1");
        // Setting testObject1 to Null will qualify it for the garbage collection
        testObject1 = null;
        // Requesting garbage collection
        System.gc();
    }
    @Override
    protected void finalize() throws Throwable
    {
        // following line will confirm the garbage collected method name
        System.out.println("Garbage collection is succcessful for " + this.myObject);
    }
}
```

Since there is no longer any reference to `testObject1` and its reference variable was made NULL, `testObject1` is no longer reachable in the code and becomes suitable for garbage collection. When the garbage collector is called, it finds `testObject1` without any reference and removes it from the heap.

The output of the code above will be as follows.

```
Garbage collection is succcessful for testObject1
```

15.6.4 Anonymous Objects

Anonymous objects can be used in Java to call methods. However, what distinguishes anonymous objects from regular objects in Java is that anonymous objects do not have any reference IDs. As per the criteria, this makes anonymous objects the perfect candidates for garbage collection.

The following code is an example of a method being used on an anonymous object:

```
package java11.fundamentals.chapter15;
public class AnonymousObjectsExample {
    public static void main(String[] args) {

        System.out.println(new AnonymousObjectsExample().myMethod());

    }
    public String myMethod() {
        return "I love this book";
    }
}
```

The output of the code above will be as follows.

```
I love this book
```

Now that you understand how anonymous objects can be used to successfully call and run methods, here is an example of how garbage collectors can be used to remove anonymous objects from the heap.

```
package java11.fundamentals.chapter15;
public class AnonymousObjectsGarbageCollectionExample {
    String myObject;
    public AnonymousObjectsGarbageCollectionExample(String myObject)
    {
        this.myObject = myObject;
    }
    public static void main(String args[])
    {
        // Anonymous Object is being initialized without a reference id
        new AnonymousObjectsGarbageCollectionExample("testObject1");
        // Requesting garbage collector to remove the anonymous object
        System.gc();
    }
    @Override
    protected void finalize() throws Throwable
    {
        // following line will confirm the garbage collected method name
        System.out.println("Garbage collection is succcessful for " + this.myObject);
    }
}
```

Since there is no any reference to the anonymous object, the garbage collector will successfully remove it from the heap. The output of the code above will be as follows.

```
Garbage collection is succcessful for testObject1
```

15.7 | JEP 318 – Epsilon: A No-Op Garbage Collector

Interesting changes were proposed to the way garbage collection was approached in Java in a March 2018 update. For Java 11 version, the goal was to develop a garbage collection mechanism that took care of memory allocation but did not quite use a significant methodology for the reclamation of reusable memory from the heap. In this update, once the heap for Java was exhausted, the JVM would shut down. this garbage collector is called no-op garbage collector, which is also known as Epsilon.

Since the proposition for the update was based on the premise that the garbage collector would not reclaim memory from the heap, the no-op garbage collector would encourage the creation of ultra-performing applications that do not have any garbage or can do without a garbage collector for memory reclamation. The no-op garbage collector is intended to be simultaneously available with other garbage collectors and will not come into effect unless it is activated explicitly.

While this comes as an interesting update for the public, developers and researchers are particularly interested in the viability and practicality of the no-op garbage collector, considering how memory allocation and garbage collection work in Java. Moreover, since the no-op garbage collector will be simultaneously available with the other garbage collectors, users will be able to benefit from the no-op garbage collector or Epsilon collector as a control variable to gauge the performance of the remaining available garbage collectors.

To test the efficiency and effect of garbage collectors on the performance and speed of an application, variable configurations of garbage collectors can also be used simultaneously with the no-op garbage collector with the same workload. By doing so, garbage collector developers will not only be able to see how the performance of applications differs based on the configuration of the garbage collector in a controlled environment, but they will also be able to understand how garbage collectors work in a more isolated manner.

The Epsilon garbage collector or no-op garbage collector can prove to be highly beneficial for a limited number of applications and libraries that do not produce any garbage. Since the presence of a garbage collector is essentially not necessary, removing the overhead of the garbage collector can improve the efficiency of the application or library that is being used. However, to create a library that supports the Epsilon garbage collector, a number of factors including the library's memory management aspect without the use of a garbage collector must be taken into consideration. Since implementation of Epsilon garbage collector will essentially leave the application with no mechanism for reclamation of memory, it will have to be ensured that garbage is either non-existent, or minimal to the extent that the memory of the application does not run out.

When talking about the Epsilon garbage collector, it is important to consider the risks and benefits of implementing a no-op garbage collector and weighing them against the problems that will manifest in order to achieve a state of no or minimal garbage in any application or program. While there are a considerable number of difficulties in reaching a no garbage state considering how memory management works in Java, a few aspects of garbage collection need to be discussed to get an idea of how achieving such a state may be possible.

There are two major mechanisms used by JVM for memory management in Java. While most memory management operations are done through the heap, the stack is equally important in order to create an application that is equal parts memory-efficient and functional. The presence and use of both the heap and stack is the primary reason why there are two different types of errors when it comes to memory management – OutOfMemoryError and StackOverflowError.

The stack is only visible and used by threads that are running at a certain point in time, that too during the execution of its particular method. When the execution of the thread that is using the stack is complete, it leaves the stack and memory is automatically freed without the use of a garbage collector. It is the memory on the heap that needs to be checked by a garbage collector to see whether or not it can be cleaned to add value to the application or program that is being used. However, it is important to note that all 8 primitive data types go directly on the stack, which makes it possible for the application or program to run efficiently without the use of a garbage collector. If the Epsilon garbage collector is to be implemented, it is suggested that primitive data types be used for the majority of purposes to ensure that there is no any additional strain or reason for the presence of a conventional garbage collector.

Contrary to popular belief, objects can also be created without the use of a garbage collector. This means that fully-functional applications and programs can still be created with the Epsilon update with just a little additional effort.

Can we avoid OutOfMemoryError?

Summary

With this discussion about the Epsilon update available, we close the topic of garbage collection and how it works in the Java language. It is assumed that the codes, examples, and scenarios provided have helped you understand the ins and outs of garbage collection in the Java language and will give you an idea of the different ways in which garbage collection can be approached. In this chapter, we have learned the following concepts:

1. How garbage collector works.
2. What is the use of garbage collection?
3. Importance of memory management.
4. How to make objects eligible for garbage collection.
5. What are the latest updates in garbage collection?

In the Chapter 16, we will learn about String and I/O operations. We will also explore Java's file management capabilities and tools available to read, write, and manipulate file content.

Multiple-Choice Questions

1. At the time of object destruction, _____ method is utilized to execute some action.
   (a) `delete()`
   (b) `finalize()`
   (c) `main()`
   (d) None of the above

2. _____ requires the highest memory.
   (a) Class
   (b) JVM
   (c) Stack
   (d) Heap
3. Where does the new object memory get allotted?
   (a) JVM
   (b) Young Space
   (c) Old Space
   (d) Young or Old Space, depending on space availability
4. _____ is a garbage collection algorithm which has two phases operation.
   (a) Space management tool
   (b) Sweep model
   (c) Cleanup model
   (d) Mark and sweep model
5. Which of the following is not a Java Profiler?
   (a) Jconsole
   (b) JVM
   (c) Jprofiler
   (d) Eclipse Profiler

Review Questions

1. How does garbage collection work in Java?
2. How do we make objects eligible for garbage collection?
3. How can you instruct the garbage collector to initiate the garbage collection process?
4. What is the inner working of the garbage collector?
5. Why is memory management important?

Exercises

1. Create a diagram that shows how objects become eligible for garbage collection.
2. Write a program that initializes a lot of objects in a loop and observe how much time it takes to crash the program.
3. Write a program that can generate stack overflow error. Document the findings.

Project Idea

Create a voting system program that can collect the entire list of the voters from all around the country and allow them to vote. This program must validate the identity of the users. Calculate the number of votes. Make sure you use good garbage collection practices so the program will not crash due to memory management issue.

Recommended Readings

1. Benjamin J. Evans, James Gough, and Chris Newland .2018. *Optimizing Java: Practical Techniques for Improving JVM Application Performance*. O'Reilly Media: Massachusetts
2. Erik Ostermueller. 2017. *Troubleshooting Java Performance: Detecting Anti-Patterns with Open Source Tools*. Apress Media: New York
3. Oracle Technetwork: Java Garbage Collection Basics – https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html